Chapitre 3

Utilisation de SAT solvers

Nous allons dans ce chapitre voir l'utilisation de SAT Solver pour résoudre des problème. Ce TP doit être fait sur **4 séances environ**.

3.1 Format CNF

Les solvers SAT utilisent des fichiers au format CNF (*Conjunctive Normal Form*). Ce sont des fichiers textes codant une formule sous forme CNF, avec les conventions suivantes :

- Les lignes commençant par `c` sont des commentaires et peuvent être placées au début du fichier.
- La première ligne du fichier (hors commentaire) est de la forme
  `p cnf x y`
  où `x` et `y` sont des entiers. L'entier `x` code le nombre de variables propositionnelles utilisées dans la formule et `y` le nombre de clauses.
- Les variables sont numérotées de 1 à `x`.
- Chaque ligne code ensuite une clause et doit se finir apr un `0`.
- Pour chaque clause (chaque ligne de clause), on met les numéros des variables apparaissant dans cette clause séparées par un espace. Si la variable est niée dans la clause, on met un moins devant.

Exemple, considérons la formule en CNF

$$(x_1 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_4).$$

Cette formule contient trois clauses et quatre variables. La première ligne sera donc
`p cnf 4 3`
Ensuite, chaque clause est codée. On obtient à la fin :
On obtient le fichier

```
p cnf 4 3
1 -4 0
2 3 4 0
-1 -4 0
```

3.2 Cyrptominisat

`Crytominisat`[1] est un solver SAT libre. Il en existe d'autres, comme `MiniSat`[2].
`Cryptominisat` est installé dans les salles TP.
Pour l'utiliser, il suffit d'exécuter la commande :

1. https://www.msoos.org/cryptominisat5/
2. http://minisat.se/

```
cryptominisat5 --verb=0 fichier.cnf
```

Si on l'utilise sur l'exemple de la section3.1 codant

$$(x_1 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_4),$$

on obtient le résultat

```
s SATISFIABLE
v -1 2 -3 -4 0
```

Cela signifie que la formule est satisfiable, avec l'instanciation :

$$x_1 = 0 \quad x_2 = 1 \quad x_3 = 0 \quad x_4 = 0.$$

Si le numéro d'une variable apparaît positivement, son instanciation est à 1, sinon son instanciation est à 0.

L'argument `-verb=0` permet d'avoir un résultat compact. L'argument `-maxsol=xx` permet, si c'est possible, d'obtenir `xx` solution différentes aux problèmes.

Q. 1 : Exécuter `Cryptominisat` sur un fichier `.cnf`.

Q. 2 : Que se passe-t-il si l'on utilise pas l'argument `-verb=0` ?

Q. 3 : Tester l'argument `-maxsol`.

3.3 Sudoku

On va utiliser `Cryptominisat` pour résoudre des Sodokus, en utilisant par exemple le codage vu en cours.

Q. 1 : Rappeler le rôle de la variable $X_{i,j,k}$ utilisée.

Q. 2 : Combien y a-t-il de variables ?

Q. 3 : Écrire une fonction de Python de linéarisation des indices des variables, c'est-à-dire qui à tout triplet (i, j, k) associe un entier de façon injective (deux triplets n'ont pas le même indice). Il est aussi préférable de ne pas utiliser d'entier supérieur au nombre de variables.

Q. 4 : Écrire une fonction Python qui effectue le codage inverse, c'est-à-dire qui à partir d'un entier retourne le triple (i, j, k) correspondant s'il existe.

Q. 5 : Écrire une fonction Python pour générer les clauses exprimant qu'il y a au moins un entier par case.

Q. 6 : Écrire une fonction Python pour générer les clauses exprimant qu'il y a au plus un entier par case.

Q. 7 : Écrire une fonction Python pour générer les clauses exprimant les contraintes sur les lignes.

Q. 8 : Écrire une fonction Python pour générer les clauses exprimant les contraintes sur les colonnes.

Q. 9 : Écrire une fonction Python pour générer les clauses exprimant les contraintes sur les blocs de taille 3 par 3.

Q. 10 : Écrire une fonction Python pour générer les clauses exprimant les conditions initiales.

Q. 11 : Chercher des Sodoku très difficile sur internet et les résoudre avec votre programme.

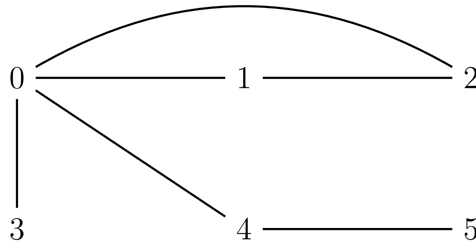
Question optionnelle (a faire chez soit si vous le souhaitez) : réaliser, sans utiliser de Solver, un programme qui résout les Sodoku. Comparer les temps de calculs (merci de partager avec votre chargé(e) de TP).

3.4 Problèmes sur les graphes

Un graphe orienté est donné par un ensemble fini de sommets (ici numéroté de 1 à n), ainsi qu'un ensemble d'arètes, qui sont des couples de sommets. On s'intéresse ici aux problèmes sur les graphes vus dans le TD2.

Les graphes seront codé en Python à l'aide d'une liste. Les sommets seront toujours numéroté de 0 à $n-1$. On aura, pour les arêtes, une liste E dont l'élément $E[i]$ sera la liste des voisins du sommet i.

Par exemple, le graphe dessiné ci dessous sera codé par la liste
`[[1,2,3],[0,2],[0,1],[0],[0],[4]]`



Q. 1 : Encire une fonction qui étant donné un graphe (par une liste) et deux entiers i et j teste s'il y a une arête entre i et j.

Q. 2 : Écrire une fonction qui étant donnée un graphe (par une liste) retourne le nombre d'arêtes qu'il contient.

Q. 3 : Écrire une fonction qui retourne la liste des voisins d'un sommet donné.

Q. 4 : Écrire une fonction qui étant donné une liste de sommets teste si cela correspond à un chemin du graphe.

Q. 5 : Coder le problème 3-COLOR pour résolution d'un solver SAT (en s'aidant du TD).

Q. 6 : Coder le problème DOMINATING SET pour une résolution par SAT solver (en s'aidant du TD).

Q. 7 : Coder le problème HAMILTONIEN pour une résolution par SAT solver (en s'aidant du TD).

Q. 8 : Coder l'algorithme de résolution 2-SAT (on pourra s'aider de la bibliothèque `igraph` pour le calcul des composantes fortement connexes).